Dokazovanje pravilnosti programov

Kako vemo, ali program deluje pravilno? Kako vemo, kakšen program želimo sestaviti?

Ločimo med **specifikacijo** in **implementacijo** programa:

- **Specifikacija** je opis, kaj naj želeni program počne.
- **Implementacija** je konkreten program, ki počne to, kar zahteva specifikacija.

Specifikacija je lahko podana bolj ali manj natančno, v človeškem jeziku ali zapisana v formalnem matematičnem jeziku. Zakaj potrebujemo specifikacije? Nekateri odgovori:

- da pridobimo opis programa, ki naj bi ga sestavili
- da lahko preverimo, ali je implementacija pravilna
- zagotovimo kompatibilnost med različnimi deli programske opreme

Danes bomo spoznali le majhen košček specifikacij, t.i. Hoarovo logiko, s katero izražamo dejstva o programih v ukaznem programskem jeziku in dokazujemo njihovo pravilnost.

Hoarova logika

V Hoarovi logiki pišemo *Hoarove trojice*

`{ P } c { Q }`

kjer sta `P` in `Q` logični formuli in `c` ukaz. Formuli `P` pravimo *predpogoj* (angl. *precondition*), formuli `Q` pravimo *končni pogoj* (ang. *postcondition*). V resnici poznamo dve različici trojic:

Delna pravilnost

`{ P } c { Q }`

pomeni

> Če velja `P` in če bo ukaz `c` končal, potem bo veljal `Q`

Popolna pravilnost

`[ P ] c [ Q ]`

pomeni

> Če velja `P`, potem se bo `c` končal in veljal bo `Q`.

Zapomnimo si: delna pravilnost ne zagotavlja, da se bo `c` končal, popolna pravilnost to zagotavlja.

Primer 1

Program `c` zamenja vrednosti spremenljivk `x` in `y`:

`{ x = m ∧ y = n } c { x = n ∧ y = m }`

Tu moramo predpostaviti, da sta `m` in `n` *duhova* (angl. ghost variable), se pravi spremenljivki, ki se ne pojavljata v `c`.

Primer 2

Program `c` poskrbi, da je `x` manjši ali enak `y`:

```
{ true } c { x ≤ y }
```

Ali znamo zapisati tak program? Da, na primer

```
x := 0 ; y := 1
```

Specifikacija to dovoli. Verjetno smo hoteli v resnici tole:

```
{ x = m ∧ y = n } c { x = min(m, n) ∧ y = max(m, n) }
```

Ko delamo s Hoarovo logiko, običajno pišemo pogoje in kodo navpično, da lahko med vrstice vrivamo pogoje.

```
{ x = m ∧ y = n }
c
{ x = min(m, n) ∧ y = max(m, n) }
```

Seveda potrebujemo nekakšna pravila sklepanja.

Pravila sklepanja

Za Hoarovo logiko veljajo naslednja pravila sklepanja.

Splošna pravila

Vedno smemo uporabiti veljavno logično in matematično sklepanje, na primer:

```
P' ⇒ P    { P } c { Q }      Q ⇒ Q'
──────────────────────────────────
          { P' } c { Q' }


{ P₁ } c { Q₁ }     { P₂ } c { Q₂ }
──────────────────────────────────
     { P₁ ∧ P₂ } c { Q₁ ∧ Q₂ }
```

Naj bodo `FV(P)` vse spremenljivke, ki se pojavljajo v formuli `P` (free variables) in `FA(c)` vse spremnljivke, ki jih `c` nastavlja (assigned variables). Na primer:

```
FV(x ≤ y ∨ x > 0) = {x, y}

FA(if x < y then x := y + 3 else skip end) = { x }
```

Velja pravilo:

```
FV(P) ∩ FA(c) = ∅
─────────────────
  { P } c { P }
```

Pravilo za `skip`

```
─────────────────
{ P } skip { P }
```

Pravilo za pogojni stavek

$$\frac{\{ P \wedge b \}\ c_1\ \{ Q \} \qquad \{ P \wedge \neg b \}\ c_2\ \{ Q \}}{\{ P \}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}\ \{ Q \}}$$

Pravilo za c_1 ; c_2

$$\frac{\{ P \}\ c_1\ \{ Q \} \qquad \{ Q \}\ c_2\ \{ R \}}{\{ P \}\ c_1\ ;\ c_2\ \{ R \}}$$

Pravilo za zanko `while`

$$\frac{\{ P \wedge b \}\ c\ \{ P \}}{\{ P \}\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ \{ \neg b \wedge P \}}$$

Formuli `P` pravimo *invarianta* zanke `while`.

Pravilo za prirejanje

$$\frac{}{\{ P[x \mapsto e] \}\ x := e\ \{ P \}}$$

Zapis `P[x ↦ e]` pomeni "v izjavi `P` zamenjaj `x` z `e`"

Popolna pravilnost

Vsa zgornja pravila, razen dveh, lahko predelamo v popolno pravilnost, na primer:

$$\frac{[P \wedge b]\ c_1\ [Q] \qquad [P \wedge \neg b]\ c_2\ [Q]}{[P]\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}\ [Q]}$$

Prva izjema je pravilo

$$\frac{FV(P) \cap FA(c) = \varnothing}{\{ P \}\ c\ \{ P \}}$$

TO JE NAROBE:

$$\frac{FV(P) \cap FA(c) = \varnothing}{[P]\ c\ [P]}$$

ker bi lahko vzeli npr.:

```
[ x > 0 ] while true do skip done [x > 0]
```

ki ga predelamo takole

$$\frac{FV(P) \cap FA(c) = \varnothing \qquad [R]\ c\ [Q]}{[R \wedge P]\ c\ [Q \wedge P]}$$

Pri zanki `while` zagotovimo, da se bo končala, tako da poiščemo količino, ki se zmanjšuje, a se ne more zmanjševati v nedogled. Na primer, to je lahko celoštevilska pozitivna vrednost.

Pozor: realna pozitivna vrednost se lahko zmanjšuje v nedogled:

```
0.1 > 0.01 > 0.001 > 0.0001 > ...
```

Tudi celoštevilska vrednost se lahko zmanjšuje v nedogled:

```
2 > 1 > 0 > -1 > -3 > -5 > -7 > ....
```

Pravilo za popolno pravilnost `while` se glasi:

Naj bo `e` količina, ki se ne more v nedogled zmanjševati (na primer naravno število):

$$\frac{[\ P \wedge b \wedge e = z\ ]\ c\ [P \wedge e < z\ ]}{[\ P\ ]\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ [\ \neg b \wedge P\ ]}$$

V tem pravilu je `z` duh. Kako pa ta pravila v praksi uporabljamo? Poglejmo nekaj primerov.

Primeri

Primer 1

Zapiši s Hoarovo logiko:

1. Program `c` se ne ustavi.
2. Program `c` se ustavi.

Primer 2

Dokaži pravilnost programa:

```
{ x ≤ 7 }
x := x + 3
{ x ≤ 10 }
```

Primer 3

Dokaži pravilnost programa:

```
{ x ≤ y }
s := (x + y) / 2
{ x ≤ s ≤ y }
```

Primer 4

Dogovor: namesto `P ∧ Q` pišemo tudi `P, Q`

Dokaži pravilnost programa:

```
[ b ≥ 0 ]
i := 0 ;
p := 1 ;
```

```
while i < b do
   p := p * a ;
   i := i + 1
done
[ p = a ^ b ]
```

Rešitev:

```
{ b ≥ 0 }
i := 0 ;
{ b ≥ 0, i = 0 }
p := 1 ;
{ b ≥ 0, i = 0, p = 1 } # logično sklepamo, je zelo easy
{ p = a ^ i, i ≤ b }
while i < b do
   { i < b, p = a ^ i, i ≤ b }
   # iz p = a^i sledi p·a = a^(i+1)
   # iz i < b sledi i+1 < b+1 sledi i+1 ≤ b (ker i, b celi števili)
   { p · a = a ^ (i + 1), (i + 1) ≤ b }
   p := p * a ;
   { p = a ^ (i + 1), (i + 1) ≤ b }
   i := i + 1
   { p = a ^ i, i ≤ b }
done
{ i ≥ b, p = a ^ i, i ≤ b } # očitno
{ i = b, p = a ^ i } # očitno
{ p = a ^ b }
```

Popolna pravilnost: zmanjšuje se celoštevilska količina `e = b - i ≥ 0`. Imamo invarianto `Q ≡ (p = a ^ i ∧ i ≤ b)`

```
while i < b do
   [ Q, i < b, b - i = z ]
   [ i < b, b - i = z ]
   p := p * a ;
   [ i < b, b - i = z ]
   ⇒
   [ b - i = z ]
   ⇔
   [ b = z + i ]
   ⇒
   [ b < z + i + 1 ]
   ⇔
   [ b - i - 1 < z ]
   ⇔
   [ b - (i+1) < z ]
   i := i + 1
   [ b - i < z ]
done
```

Primer 5

Dokaži pravilnost programa:

```
[x = m ∧ y = n]
if y < x then
   x := x + y ;
   y := x - y ;
   x := x - y
else
   skip
```

```
end
[ x = min(m, n) ∧ y = max(m, n) ]
```

Rešitev:

```
[x = m ∧ y = n]
if y < x then
   [ y < x, x = m ∧ y = n]
   [ n < m, x = m ∧ y = n]
   [ y = n = min(m, n) ∧ x = m = max(m, n) ]
   [ (x+y)-((x+y)-y) = min(m, n) ∧ ((x+y)-y) = max(m, n) ]
   x := x + y ;
   [ x-(x-y) = min(m, n) ∧ (x-y) = max(m, n) ]
   y := x - y ;
   [ x-y = min(m, n) ∧ y = max(m, n) ]
   x := x - y
   [ x = min(m, n) ∧ y = max(m, n) ]
else
   [x ≤ y, x = m ∧ y = n]
   [m ≤ n, x = m ∧ y = n]
   skip
   [ m ≤ n, x = m ∧ y = n ] # očitno sledi
   [ x = min(m, n) ∧ y = max(m, n) ]
end
[ x = min(m, n) ∧ y = max(m, n) ]
```